Programación avanzada en Python

Lección 6: Paquetes y módulos

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Paquetes y módulos

PRESENTACIÓN Y OBJETIVOS

En este sexto capítulo estudiaremos cómo trabajar con paquetes y módulos. Primero empezaremos aprendiendo cómo buscar paquetes y seguidamente veremos cómo crear nuestros propios módulos y paquetes.

Objetivos

En esta lección aprenderás a:

- Que es la programación modular
- Buscar e instalar paquetes
- Crear modulos y paquetes

1. PROGRAMACIÓN MODULAR

Los módulos y paquetes de Python definen a la programación modular. La programación modular divide los grandes programas o aplicaciones en un número de pequeñas unidades y los hace reutilizables.

- Los módulos son fáciles de manejar y pueden centrarse fácilmente en la corrección de errores. Si un programa grande se divide en partes más pequeñas, es realmente útil para depurar cada sección.

- Un módulo puede mantener y hacer cambios fácilmente sin conocer todas las partes del programa. Los módulos pueden distinguir sus funciones bajo diferentes nombres para poder ser accedidos fácilmente.

- Una función o atributos bien definidos pueden ser reutilizados en cualquier lugar especificando su nombre y características evitando la redundancia en los programas.

Módulos

Los módulos son secciones que contienen simplemente una o dos funciones que realizan cualquier tarea. Un módulo puede ser definido como un programa Python y puede importarse en otros programas.

Paquetes

Los paquetes son una colección de módulos. Estos son ampliamente utilizados en los programas de Python. Los paquetes pueden almacenar muchos módulos y también subpaquetes.

Import

Un módulo se define como un programa de Python y puede ser accedido a otros programas utilizando la palabra clave 'import'.

Por ejemplo:

Definir un pequeño módulo y guardarlo en un archivo como modulo.py y crear otro programa sample.py que utilice la función o características del módulo creado como modulo.py. Se utilizará el módulo con 'import' de la siguiente manera:

import módulo

Del mismo modo, podemos importar diferentes módulos.

Es importante que el módulo y el programa que lo utiliza estén en un mismo directorio de trabajo.

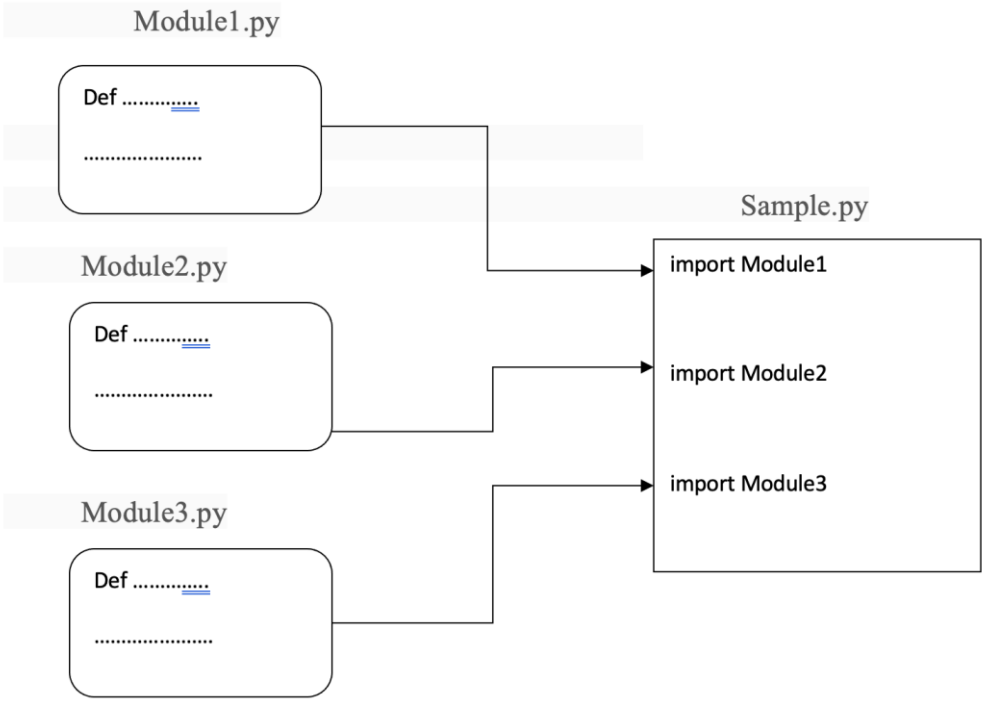


Figura 1.1: Ejemplo import

2. BÚSQUEDA E INSTALACIÓN DE PAQUETES

Los paquetes son una colección de módulos. Hay muchos paquetes incorporados en el core de Python que se utilizan ampliamente en los programas que desarrollamos.

Pip es un instalador recomendado, que ayuda a instalar y actualizar los paquetes de Python con una sola línea de comando, para comprobar su versión, ejecutamos lo siguiente:

python -m pip – versión

Podemos comprobar los paquetes ya instalados en nuestro entorno con:

pip list

Para instalar un nuevo paquete:

python -m pip install nombre_de_paquete

Un paquete ya instalado puede ser actualizado a la versión más reciente:

python -m pip install - - upgrade nombre_paquete

Hay muchas colecciones de paquetes en python para diferentes propósitos.

A continuación se muestra un listado de los utilizados categorizados por utilidades:

Paquetes de preprocesamiento de texto

`string` — Common string operations
`re` — Regular expression operations
`difflib` — Helpers for computing deltas
`textwrap` — Text wrapping and filling
`unicodedata` — Unicode Database
`stringprep` — Internet String Preparation
`readline` — GNU readline interface
`rlcompleter` — Completion function for GNU readline

Paquetes para tipos de datos

`datetime` — Basic date and time types
`zoneinfo` — IANA time zone support
`calendar` — General calendar-related functions
`collections` — Container datatypes
`collections.abc` — Abstract Base Classes for Containers
`heapq` — Heap queue algorithm
`bisect` — Array bisection algorithm
`array` — Efficient arrays of numeric values
`weakref` — Weak references
`types` — Dynamic type creation and names for built-in types
`copy` — Shallow and deep copy operations
`pprint` — Data pretty printer
`reprlib` — Alternate `repr()` implementation
`enum` — Support for enumerations
`graphlib` — Functionality to operate with graph-like structures

Módulos numéricos y matemáticos:

`numbers` — Numeric abstract base classes
`math` — Mathematical functions
`cmath` — Mathematical functions for complex numbers
`decimal` — Decimal fixed point and floating point arithmetic
`fractions` — Rational numbers
`random` — Generate pseudo-random numbers
`statistics` — Mathematical statistics functions

Acceso a archivos y directorios

`pathlib` — Object-oriented filesystem paths
`os.path` — Common pathname manipulations
`fileinput` — Iterate over lines from multiple input streams
`stat` — Interpreting `stat()` results
`filecmp` — File and Directory Comparisons
`tempfile` — Generate temporary files and directories
`glob` — Unix style pathname pattern expansion
`fnmatch` — Unix filename pattern matching
`linecache` — Random access to text lines
`shutil` — High-level file operations

Paquetes para la comprensión y el archivo de datos

`zlib` — Compression compatible with **gzip**
`gzip` — Support for **gzip** files
`bz2` — Support for **bzip2** compression
`lzma` — Compression using the LZMA algorithm
`zipfile` — Work with ZIP archives
`tarfile` — Read and write tar archive files

Paquetes para diferentes formatos de archivos

`csv` — CSV File Reading and Writing
`configparser` — Configuration file parser
`netrc` — netrc file processing
`xdrlib` — Encode and decode XDR data
`plistlib` — Generate and parse Apple `.plist` files

Paquetes para acciones genéricas del sistema operativo

`os` — Miscellaneous operating system interfaces
`io` — Core tools for working with streams
`time` — Time access and conversions
`argparse` — Parser for command-line options, arguments and sub-commands
`getopt` — C-style parser for command line options
`logging` — Logging facility for Python
`logging.config` — Logging configuration
`logging.handlers` — Logging handlers
`getpass` — Portable password input
`curses` — Terminal handling for character-cell displays
`curses.textpad` — Text input widget for curses programs
`curses.ascii` — Utilities for ASCII characters
`curses.panel` — A panel stack extension for curses
`platform` — Access to underlying platform's identifying data
`errno` — Standard errno system symbols
`ctypes` — A foreign function library for Python

Paquetes para la ejecución concurrente

`threading` — Thread-based parallelism
`multiprocessing` — Process-based parallelism
`multiprocessing.shared_memory` — Provides shared memory for direct access a
The `concurrent` package
`concurrent.futures` — Launching parallel tasks
`subprocess` — Subprocess management
`sched` — Event scheduler
`queue` — A synchronized queue class
`contextvars` — Context Variables
`_thread` — Low-level threading API

Paquetes para protocolo y soporte de Internet

`webbrowser` — Convenient Web-browser controller
`cgi` — Common Gateway Interface support
`cgitb` — Traceback manager for CGI scripts
`wsgiref` — WSGI Utilities and Reference Implementation
`urllib` — URL handling modules
`urllib.request` — Extensible library for opening URLs
`urllib.response` — Response classes used by urllib
`urllib.parse` — Parse URLs into components
`urllib.error` — Exception classes raised by urllib.request
`urllib.robotparser` — Parser for robots.txt
`http` — HTTP modules
`http.client` — HTTP protocol client
`ftplib` — FTP protocol client
`poplib` — POP3 protocol client
`imaplib` — IMAP4 protocol client
`nntplib` — NNTP protocol client
`smtplib` — SMTP protocol client
`smtpd` — SMTP Server
`telnetlib` — Telnet client
`uuid` — UUID objects according to **RFC 4122**
`socketserver` — A framework for network servers
`http.server` — HTTP servers
`http.cookies` — HTTP state management
`http.cookiejar` — Cookie handling for HTTP clients
`xmlrpc` — XMLRPC server and client modules
`xmlrpc.client` — XML-RPC client access
`xmlrpc.server` — Basic XML-RPC servers
`ipaddress` — IPv4/IPv6 manipulation library

Paquetes para el manejo de datos en Internet

`email` — An email and MIME handling package
`json` — JSON encoder and decoder
`mailcap` — Mailcap file handling
`mailbox` — Manipulate mailboxes in various formats
`mimetypes` — Map filenames to MIME types
`base64` — Base16, Base32, Base64, Base85 Data Encodings
`binhex` — Encode and decode binhex4 files
`binascii` — Convert between binary and ASCII
`quopri` — Encode and decode MIME quoted-printable data
`uu` — Encode and decode uuencode files

Paquetes para gráficos

Chaco - Creates interactive plots
gnuplot.py - Based on gnuplot
Matplotlib - Production quality output in a wide variety of formats
Plotly - Interactive, publication-quality, web based charts
PyX - Postscript and PDF output, (La)TeX integration
ReportLab includes a charting package
Veusz - Postscript output with a PyQt front end
pyqtgraph - Pure-python plotting and graphics library based on PyQt and numpy.

Paquetes para ciencia de datos y algoritmos de aprendizaje automático:

NumPy
SciPy
BeautifulSoup
Scrappy
Pandas
Matplotlib
Plotly

Seaborn
Scikit Learn
PyCaret
TensorFlow
Keras
PyTorch

3. CREANDO NUESTROS PROPIOS MÓDULOS

Un módulo puede ser definido como un programa de Python y se puede importar en otros programas.

Veamos un ejemplo con un módulo llamado module.py:

module.py

```
1 def square(n):
2   return n*n
3
4 list = [1,2,3,4]
5 string = "hello"
```

Este es un archivo module.py que contiene:

- Una función cuadrada
- Lista
- Cadena

El módulo se guarda en el directorio de trabajo actual y otro programa sample.py utiliza las funciones y atributos en el module.py. Para ello el programa sample.py debe de hacer referencia al nuevo módulo creado:

```
import module
```

Esta sentencia importará todas las funciones del módulo al programa actual.

La función y los atributos se pueden utilizar en el programa actual con un operador de punto:

nombre_del_módulo.nombre_de_la_función

nombre_del_módulo.atributo1

y así sucesivamente.

En el ejemplo anterior, el módulo puede llamar a su función y atributos como:

```
print(module.list)
```

```
[1, 2, 3, 4]
```

```
print(module.string)
```

```
hello
```

```
sq = module.square(5)
print(sq)
```

```
25
```

Declaraciones de importación

Los módulos creados son llamados con la sentencia import y hay diferentes maneras de llamar a los módulos usando import.

1. Import nombre_del_módulo

Tomemos como ejemplo el módulo anterior, llamamos al módulo por su nombre con la sentencia import. Para las variables hacemos referencia a ellas con la notación de punto.

```
import module
```

```
print(module.list)
```

```
[1, 2, 3, 4]
```

Se pueden especificar varios módulos separándolos con comas

import módulo1,módulo2,.....

2. from nombre_del_módulo import nombre/nombres

La palabra clave 'from' se puede utilizar con import para importar únicamente los objetos o funciones del módulo especificado, por lo que, no es necesario importar todo el módulo si el programa actual sólo necesita algunas funciones.

En el module.py, si sólo necesitamos la función square, podemos importar la solo la función de la siguiente manera:

```
from module import square
```

En este caso, la notación de punto no es necesaria para llamar a la función. Se puede llamar directamente:

```
print(square(5))
```

```
25
```

Podemos especificar más de un nombre de objeto en la declaración de importación:

```
from module import square,list

print(square(5))
print(list)
```

```
25
[1, 2, 3, 4]
```

También es posible importar todos los objetos del módulo con la misma sentencia especificándolo con el símbolo '*'

```
from module import *

print(list)
print(string)
print(square(25))
```

```
[1, 2, 3, 4]
hello
625
```

3. from nombre_del_módulo import nombre_alternativo

Las funciones y otros objetos que se importan del módulo se pueden representar con otros nombres simples y pueden utilizarse en todo el programa.

El nombre alternativo se especifica con la palabra clave 'as'

```
from module import square as sqr

print(sqr(2))
```

```
4
```

Del mismo modo, podemos especificar todos los objetos en una sola línea con coma

```
from module import square as sqr, list as ls, string as str

print(ls)
print(str)
print(sqr(1))
```

```
[1, 2, 3, 4]
hello
1
```

4. import nombre_del_módulo como nombre_alternativo

Los módulos pueden ser importados con otro nombre alternativo

```
import module as mod

print(mod.list)
print(mod.square(3))
```

```
[1, 2, 3, 4]
9
```

Normalmente todas las declaraciones de importación se colocan al principio de la función.

También podemos importar la función dentro de una función u otras declaraciones de condición.

Por ejemplo, la importación del módulo module.py se puede especificar dentro de una función, por lo que el módulo sólo importa cuando llamemos a la función

```
def cube(n):
  from module import square as sqr
  cube = sqr(n) * n
  return cube
```

Tabla de símbolos

Cuando se importa un módulo, pensamos que se importan directamente todas las definiciones dentro del módulo al programa. Pero lo que realmente ocurre es que se importa una tabla de símbolos.

Hay dos tipos:

Tabla de símbolos privada y tabla de símbolos local

- Tabla de símbolos privada.

  Cada módulo tiene una tabla de símbolos privada que almacena los objetos como términos globales.

- Tabla de símbolos local.

  Cuando se importa el módulo al programa, el nombre del módulo se copia en la tabla de símbolos del usuario que lo llama y esa es la tabla de símbolos local.

Función dir ()

dir () es una función incorporada que se utiliza para listar los nombres de los objetos presentes en la tabla de símbolos local actual.

Para obtener los nombres de un módulo se puede especificar el nombre del módulo como parámetro

```
import module
dir(module)
```

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'list',
 'square',
 'string']
```

Aquí contiene los objetos definidos en el módulo - list, square and string.

Es importante conocer que sucede si se pone una función de impresión en un módulo que va a ser importado, veamos un ejemplo:

module.py

```
1 def square(n):
2   return n*n
3
4 list = [1,2,3,4]
5 string = "hello"
6
7 print(list)
8 print(string)
```

Importamos el módulo:

```
import module
```

```
[1, 2, 3, 4]
hello
```

Al importar el módulo se imprimirá a sí mismo. Este comportamiento se puede evitar añadiendo una función principal:

module.py

```
1 def square(n):
2   return n*n
3
4 list = [1,2,3,4]
5 string = "hello"
6
7 if (__name__ == '__main__'):
8   print(list)
9   print(string)
10
```

De esta manera no imprimirá los objetos especificados en la función print y sólo imprimirá cuando se han llamados.

4. CREAR NUESTROS PROPIOS PAQUETES

Cuando se crean muchos módulos, se vuelve más confuso almacenar y acceder en función de su uso. Los paquetes ayudan a agrupar los métodos que tienen la misma funcionalidad juntos.

Los módulos con la misma funcionalidad o para el mismo propósito se pueden guardar bajo un directorio y nombre, el cual será el nombre del paquete.

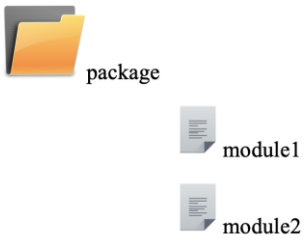


Figura 4.1 Ejemplo agrupación módulos

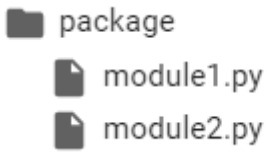
Aquí el directorio, package es el paquete y module1 y module2 son los módulos dentro del paquete.

El módulo dentro del paquete puede ser accedido de la siguiente manera:

Nombre_del_paquete.nombre_del_módulo

Veamos un ejemplo

Creamos un directorio package y almacenamos los archivos python modulo1.py y modulo2.py



Definimos los ficheros Python con funciones:

module1.py

```
1 def mod1():
2   print("mod1")
```

module2.py

```
1 def mod2():
2   print("mod2")
```

Llamamos a las funciones con la sentencia *import*:

```
import package.module1, package.module2
package.module1.mod1()
package.module2.mod2()
```

```
mod1
mod2
```

También podemos importar los módulos de diferentes maneras:

```
from package.module1 import mod1
mod1()
```

```
mod1
```

```
from package.module2 import mod2 as m2
m2()
```

```
mod2
```

El paquete puede ser importar directamente:

```
import package
```

Pero no puede acceder al módulo directamente como package.modulo1, para ello, los módulos deben ser inicializados mediante la inicialización del paquete.

Inicialización del paquete

Al igual que la función __init__ dada en la clase es utilizada para la inicialización, aquí el archivo __init__.py se utiliza para inicializar los módulos del paquete.

El archivo se define dentro del directorio del paquete al principio, antes de todos los archivos de los módulos.

Cuando se importa el paquete, el archivo init se invoca automáticamente.

Figura 4.2 Ejemplo __init__

Veamos un ejemplo:

Primero definimos el archivo init:

__init__.py

```
1 print("this is __init__ of pakcage")
2 list = [1,2,3,4]
```

Cuando importamos el paquete:

```
import package
```

```
this is __init__ of pakcage
```

Ahora podemos acceder directamente a la lista:

```
package.list
```

```
[1, 2, 3, 4]
```

De la misma manera, el archivo __init__ también puede importar los módulos, lo cual nos permitirá acceder directamente a través del import en el programa que sea importado:

__init__.py

```
1 print("this is __init__ of pakcage")
2 import pakcage.module1, pakcage.module2
```

Así que modificando el archivo __init__ con la declaración de importación que importa los módulos 1 y 2, nos permite llamar a los módulos en el programada de destino:

```
import package
package.module1.mod1()
```

```
mod1
```

Comprobemos la función *dir ()* del paquete

```
import package
dir(package)
```

```
'__doc__',
'__file__',
'__loader__',
'__name__',
'__package__',
'__path__',
'__spec__',
```

No muestra los módulos que importamos, para ello se debe especificar en el archivo __init__ lo siguiente:

__init__.py

```
1 __all__ = [
2   'module1',
3   'module2'
4 ]
```

Entonces si volvemos a comprobar la función dir:

```
import package
dir(package)
```

```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 'module1',
 'module2']
```

Sub paquetes

Podemos añadir paquetes dentro del paquete como:

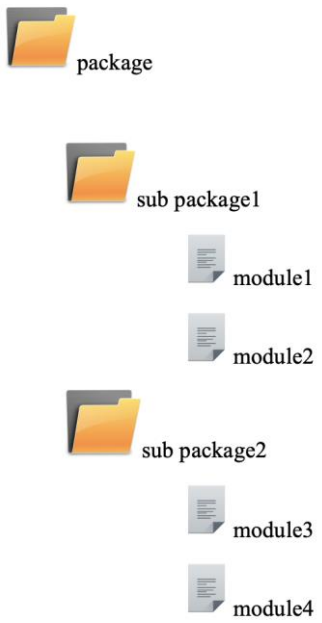


Figura 4.3: Ejemplo Sub paquetes

Podemos acceder al módulo de la siguiente manera:

nombre_paquete.nombre_subpaquete.nombre_módulo

Veamos un ejemplo con 4 módulos:

module1.py
```
1 def mod1():
2   print("mod1")
```

module2.py
```
1 def mod2():
2   print("mod2")
```

module3.py
```
1 def mod3():
2   print("mod3")
```

module4.py
```
1 def mod4():
2   print("mod4")
```

Ahora importamos los módulos con la sintaxis mencionada anteriormente.

```
import package.sub_pack1.module1
```

```
package.sub_pack1.module1.mod1()
```

```
mod1
```

Del mismo modo, podemos acceder al sub paquete 2

```
import package.sub_pack2.module3
import package.sub_pack2.module4

package.sub_pack2.module3.mod3()
package.sub_pack2.module4.mod4()
```

```
mod3
mod4
```

5. PUNTOS CLAVE

- Un módulo se define como un programa de Python y puede ser accedido a otros programas utilizando la palabra clave **import**.
- **Pip** es un instalador recomendado, que ayuda a instalar y actualizar los paquetes de Python con una sola línea de comando. Para comprobar la versión de pip ejecutamos lo siguiente
- Los **paquetes** son una colección de módulos. Hay muchos paquetes incorporados en el core de Python que se utilizan ampliamente en los programas que desarrollamos.
- Un **módulo** puede ser definido como un programa de Python y se puede importar en otros programas.